A Comparative Study of the Locality Characteristics of an Object-Oriented Language

A Thesis

Presented to the

Department of Computer Science

Brigham Young University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

by

Mark K. Gardner

November 1994

This thesis by Mark K. Gardner is accepted in its present form by the Department of Computer Science of Brigham Young University as satisfying the thesis requirement for the degree of Master of Science.

Aurel Cornell, Committee Chair

Scott N. Woodfield, Committee Member

Date

David W. Embley, Graduate Coordinator

Acknowledgements

I am grateful to my advisors, Aurel Cornell and Scott Woodfield, for their day to day encouragement and support. I am also grateful for the assistance and advice of Evan Ivie, Kelly Flanagan, and Greg Thompson. I am especially thankful for the support of my loving wife, Laurinda, and my daughters, Melanie and Diane. It has been through their love, support, and tolerance that I have been able to achieve this goal. Finally, I would like to thank the Creator of All for making the world an interesting place and endowing me with an insatiable curiosity.

Table of Contents

Chapter		Page
	Acknowledgements	iii
	Table of Contents	iv
	List of Illustrations	v
	List of Tables	v
1	Introduction	1
	1.1 Purpose of Thesis	2
	1.2 Locality of Reference	2
2	Experimental Method	4
	2.1 Test Programs	4
	2.1.1 The C Program	5
	2.1.2 The C++ Program	6
	2.1.3 Program Input	9
	2.2 Trace Acquisition	11
	2.3 Analysis Technique	13
3	Experimental Results	17
	3.1 Trace Statistics	17
	3.2 Cache Miss Rates	22
4	Conclusions	29
Appendices		31
	A1 Annotated Bibliography	32
	A2 Availability of Source Code and Traces	40

List of Illustrations

Figure		Page
1	ORM of Oberon symbols	7
2	Determining the class of a specialization	8
3	The hardware data acquisition system	12
4	Miss rates for direct-mapped caches with 32 byte lines	23
5	Miss rates for 2-way set associative caches with 32 byte lines	24
6	Access multiplier vs latency ratio (direct-mapped, 32 byte lines)	26
7	Access multiplier vs latency ratio (2-way set associative, 32 byte lines)	27

List of Tables

Table		Page
1	Files used as input to the test programs	10
2	Representative state of the art caches	15
3	Representative system latency ratios	16
4	User and OS statistics for the traces	18
5	Traces statistics for the C program	19
6	Traces statistics for the C++ program	20
7	Mean cache miss rates in percent	22
8	Access multipliers corresponding to figures 6 and 7	25

Chapter 1

Introduction

The past five years have witnessed wide spread adoption of the object-oriented paradigm as the preferred method of software development. Some of the most widely touted benefits include support for better abstraction, better reuse of software, and improved maintenance. While not everyone agrees that object-oriented technology lives up to the hype, most do agree that the process of developing software is enhanced by adopting an object-oriented viewpoint. There is a price to be paid, however. It appears that object-oriented programs execute more slowly and consume more memory.

Such a state of affairs is not without precedent even within a discipline as young as computing. The switch from assembly to high-level languages brought an increase in productivity, but at a price. In return for more productivity, the programmer had to sacrifice control over certain details, such as how computations are mapped to machine instructions. At first, the result was a reduction in performance. However as compiler technology improved, much of the performance was regained. Now few consider writing a large program exclusively in assembly language. Likewise, it may well be that future programmers will do the vast majority of their programming in an object-oriented language. For this to occur, however, more than anecdotal evidence of reduced performance needs to be obtained and the causes of reduced performance must be identified and corrected.

1.1 Purpose of the Thesis

This study was undertaken to investigate the hypothesis that the performance of an object-oriented program, as measured by its locality of reference, is less than that of a comparable non-object oriented program. An additional goal was to generate accurate memory reference traces to be used in further research.

1.2 Locality of Reference

Locality of reference is a measure of the degree to which a program clusters accesses to information while executing. Clustering with respect to time is called temporal locality, while clustering with respect to space is called spatial locality. Good temporal and spatial locality allow optimizations which increase computer performance.

As an example, consider the memory hierarchy typical of modern computer systems. Such systems often contain instruction and data caches which store the most frequently used information so that the processor seldom has to wait for main memory. Also, most modern operating systems provide virtual memory support which allows processes to be larger than physical memory. Even disk controllers contain caches to reduce the time it takes to retrieve information from secondary storage. In each case, exploiting the locality of reference exhibited by programs allows the performance of the system to be improved. Without locality of reference, current levels of performance could only be obtained at a much greater cost.

Experimental evidence that programs exhibit locality of reference began in the mid 1960's. The work of Hatfield on program access patterns[1], Denning on the working sets

[1] D. Hatfield, "Experiments on Page Size, Program Access Patterns, and Virtual Memory Performance", *IBM Journal of Research and Development*, Vol. 16, No. 1, January 1972, pp. 58-66.

of processes[2], and Ferrari et. al. on restructuring programs[3] represent early results. Literature dealing specifically with object-oriented technology has concentrated on the paging behavior of Smalltalk-80 systems running on virtual memory machines. Stamos has shown that the locality of a Smalltalk system can be significantly improved by statically reordering the placement of objects in virtual memory pages[4]. Williams et. al. have reported that the locality of a Smalltalk system can be further improved by using a dynamic placement algorithm[5]. These studies suggest that even with restructuring, the paging performance of object-oriented applications still lags that of non-object oriented systems.

Recently, Calder et. al. reported that the execution characteristics of a number of C++ programs showed less locality than similar C programs[6]. Although this thesis also compares C and C++ locality characteristics, there are some of significant differences between it and the work in reference 6. One purpose of this study was to generate traces for use in further research. As a result, it was necessary to insure that the tasks performed were comparable. Thus, this study compares two programs which were specifically designed to perform the same task, while the previous work compared pairs of programs which performed similar but not identical tasks. This study also used hardware rather than software for trace acquisition and estimates the effect of differences in locality on system performance.

---

2 P. Denning, "The Working Set Model for Program Behavior", *Communications of the ACM*, Vol. 11, No. 5, May 1968, pp. 323-333.

3 D. Ferrari and M. Kobayashi, "Program Restructuring Algorithms for Global LRU Environments", *Proceedings of the International Computing Symposium*, 1977, pp. 277-283.

4 J. Stamos, "Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory", *ACM Transactions on Computer Systems*, Vol. 2, No. 2, May 1984, pp. 155-180.

5 I. Williams, M. Wolczko, T. Hopkins, "Dynamic Grouping in an Object Oriented Virtual Memory Hierarchy", *Proceedings of 1987 European Conference on Object-Oriented Programming*, Springer-Verlag, Paris, Vol. 276, pp. 79-88.

6 B. Calder, D. Grunwald, and B. Zorn, "Quantifying Behavioral Differences Between C and C++ Programs", TR698, University of Colorado, January 1994.

Chapter 2

Experimental Method

2.1 Test Programs

Since the purpose of this study was to compare the locality behavior of an object-oriented program with a non-object oriented program, a search of the Internet was conducted to find a program of each type that performed the same task. No such pair was found. Thus, it was necessary to write at least one program to insure that differences in the computational task would not influence the result.

One of the goals in selecting or writing a pair of programs was the desire that they represent a "real world" application performing a "real world" task. At the same time, the task needed to be small enough that writing the programs did not consume an inordinate amount of time. The task selected was that of compiling source code for the Oberon language. The language is small and a model implementation of the compiler has been published[7]. Sections 2.1.1 and 2.1.2 further discuss the two programs.

The choice of implementation languages required careful consideration. Previous studies have used Smalltalk-80 for experimentation because of its flexibility and pure object-oriented approach. However, there appears to be few investigations which have chosen more traditional compiled languages. For this reason, the C and C++ languages were chosen for use in this study, both because of their similarity and because of their wide spread use in industry. An additional benefit of this choice is the existence of compilers which compile both languages. Having a common compiler reduces the possibility that

---

7 N. Wirth and J. Gutknecht, *Project Oberon: The Design of an Operating System and Compiler*, Addison-Wesley ACM Press, New York, New York, 1992.

differences in the result are due to differences in the code generation and optimization engines.

Although there are many compilers that process both C and C++, the GNU gcc v2.6.0 compiler was chosen (primarily for economic reasons as the test hardware had neither an ANSI C nor a C++ compiler installed). The GNU loader from binutils v2.4 was also installed because the loader supplied by the hardware vender had troubles with the C++ object files. The GNU C++ library, libg++ v2.6, was compiled to support the C++ program. All other libraries and tools were those distributed with the operating system.

2.1.1 The C Program

The C program is a direct translation of the source code for the Oberon compiler created by Niklaus Wirth. Although a detailed analysis of the program is beyond the scope of this work, a general description of the design and implementation will allow a better discussion of the C++ version in the next section.

The published Oberon compiler generates machine code for the Oberon environment[8] running on a Ceres workstation with a National Semiconductor 32000 CPU. The compiler is written in Oberon and employs a recursive descent approach to parsing. The basic data structures are simple and the style is typical of current programming practice. Except for the need to modify the code to take into account the differences in byte ordering (the original code was for a little-endian machine while the test machine is big-endian), the translation was straight forward.

---

8 The Oberon environment is available by anonymous ftp from neptune.inf.ethz.ch and can be installed on several popular platforms. The source code for the system as documented in *Project Oberon: The Design of an Operating System and Compiler* is also available from the same site.

2.1.2 The C++ Program

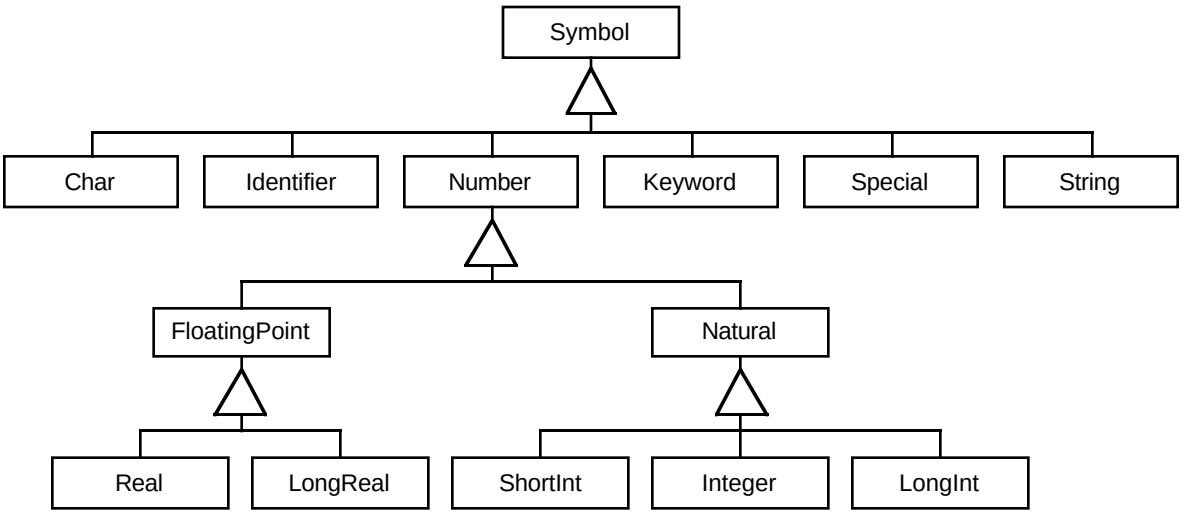
In keeping with the goals of this study, an attempt was made to exaggerate the potential differences in locality that occur from applying an object-oriented approach over a more traditional one. Thus as a design objective, the C++ program makes full use of object-oriented techniques as much as possible. The example of the Smalltalk-80 language and environment was followed in this regard. Whenever two classes of objects were conceptually distinct, two classes were created even if the class would only have a single member. In addition, all methods were made virtual in order to accentuate the effects of polymorphic dispatch. This design philosophy resulted in a proliferation of classes and objects. Such adherence to a pure object-oriented approach may be unwise in a commercial C++ environment, but was done to exaggerate the reduction in locality resulting from the adoption of the object-oriented viewpoint. It should also be noted that, like Smalltalk-80, only single inheritance was used in spite of the capability of multiple inheritance in C++.

As an example of the proliferation of classes, consider the various types of symbols shown in the OSA Object Relationship Model (ORM)[9] of figure 1. The specializations of the Number object class were created because each number requires a different data representation. Contrast this with a typical design in which a single Symbol object class would be created and variant record type used to distinguish between the actual contents of the specializations.

---

9 In an OSA Object Relationship Model, rectangles represent classes of objects, while the lines between object classes represent the sets of relationships between objects belonging to those classes. The triangular symbols on the lines between object classes indicates a generalization/specialization relationship. In this diagram, the Integer class is a specialization of the Natural class, which is a specialization of the Symbol class. For more information about OSA, see *Object-Oriented Systems Analysis: A Model-Driven Approach* by Embley, Kurtz, and Woodfield.

Figure 1: ORM of Oberon symbols



Note that the although the Special and Keyword object classes could have been specialized further for each special symbol or specific keyword, instances of those object classes used a tag attribute to specify the specialization instead. This was motivated by the fact that no additional data was needed by a specialization.

In this design, the scanner can preserve the necessary information about the symbols read from the input file in specializations of the Symbol class. However, the compiler must know what kind of symbol the instance represents (i.e., to which class does the specialization belong) in order to access that information and continue the compilation process. This was done by enumerating each of the possible specializations in the head of each generalization/specialization relationship. A polymorphic method is also defined for the class and each specialization is required to redefine that method to return the correct enumeration. The compiler calls the method and acts upon the result as appropriate. This technique is shown in the code fragment of figure 2.

Figure 2: Determining the class of a specialization

```
class Symbol {
   public:
      enum SymbolClass {
         Undefined, Number, String, Identifier, Special, Keyword, Char
      };
      virtual SymbolClass GetSymbolClass(void) const;
};

Symbol::SymbolClass Symbol::GetSymbolClass(void) const {
   return (Undefined);
}

class Keyword : public Symbol {
   public:
      enum KeywordClass {
         Undefined, DIV, MOD, OR, ...
      };
      Keyword(KeywordClass keywordClass);
      virtual SymbolClass GetSymbolClass(void) const;
      virtual KeywordClass GetKeywordClass(void) const;
   private:
      KeywordClass keyword;
};

Symbol::SymbolClass Keyword::GetSymbolClass(void) const {
   return (Symbol::Keyword);
}

Keyword::KeywordClass Keyword::GetKeywordClass(void) const {
   return (keyword);
}

/* As an example... */
if (nextSymbol->GetSymbolClass() == Symbol::Keyword &&
   ((Keyword *) nextSymbol)->GetKeywordClass() == Keyword::OR) {
   /* do something */
}
```

The Oberon language relies upon a garbage collector to free objects that have been dynamically allocated on the heap. C and C++, on the other hand, leave the reclamation of memory up to the programmer. Such differences in philosophy lead to rather different styles of programming. For this study, the garbage collection style was assumed. Although, the type of garbage collector used can have a big impact on the locality of the

program[10], once garbage collection is assumed the style of the program does not vary significantly. Of the possible types of garbage collectors, the "null collector" was selected as the most simple. When objects are no longer needed, they are simply ignored and the space they occupy is not reclaimed until the program terminates. This results in larger memory requirements for the program.

It is expected that not reclaiming objects will lead to increased heap dilution and a reduction in the locality of reference. Since both programs employ a null collector, differences in locality caused by dilution will only become evident if there is a vast difference in the number of objects allocated on the heap. The C++ program allocates objects to excess (while the C version does not) and therefore should show additional impact due to heap dilution. This is in keeping with the attempt to exaggerate the effect of adopting an object-oriented approach.

Note, the choice of implementation algorithms is likely to have a large effect on locality. As the thrust of this study was not a comparison of algorithms, the C++ program employed the same algorithms as the C program. For example, both the C and C++ programs used a recursive decent approach to parsing. However, the C++ program used a hierarchy of classes representing constructs from the grammar, rather than a set of recursive procedures. Instances of these classes were created and methods called as appropriate to parse the input.

2.1.3 Program Input

Memory access characteristics of programs are influenced by the input files processed. Thus, the input files used in this study should be of varying lengths and complexity. The source files chosen were those files from the Oberon environment necessary to implement the compiler itself. These included some of the source files for the operating system. While the characteristics of the input files were not thoroughly investigated, nearly every construct of the Oberon language is represented.

---

[10] R. Courts, "Improving Locality of Reference in a Garbage-Collecting Memory Management System", *Communications of the ACM*, Vol. 31, No. 9, September 1988, pp 1128-1138.

Table 1: Files used as input to the test programs

File Name	Classification	Lines	Bytes
Compiler.Mod	Compiler	971	30,006
Display.Def†	Operating System	87	2,104
FileDir.Mod	Operating System	368	11,264
Files.Mod	Operating System	449	13,382
Fonts.Mod	Operating System	118	3,163
Input.Mod	Operating System	70	2,079
Kernel.Def†	Operating System	41	1,166
MenuViewers.Mod	Operating System	226	7,947
Modules.Mod	Operating System	229	7,317
OCC.Mod	Compiler	611	19,056
OCE.Mod	Compiler	975	32,009
OCH.Mod	Compiler	564	18,256
OCS.Mod	Compiler	317	9,440
OCT.Mod	Compiler	590	19,427
Oberon.Mod	Operating System	488	13,218
Reals.Def†	Operating System	39	778
TextFrames.Mod	Operating System	867	31,141
Texts.Mod	Operating System	842	26,606
Viewers.Mod	Operating System	248	7,334
Total		8,100	255,693

† Definitions only, the separate implementation is in assembly language

Table 1 presents a summary of the files used as input to the two programs. Since the prefix of each file name is unique, only the prefix will be used when referring to the input files throughout the rest of the study.

2.2 Trace Acquisition

Memory reference traces can be acquired by software or hardware means. Each has its advantages and disadvantages. In the software approach, instructions are inserted into the instruction stream to log memory references[11, 12, 13]. These instructions can be generated by a modified compiler or linker, or by special post-processing of the executable. The primary advantage of the software approach is greater flexibility and relatively low cost. The disadvantages are slow execution (large time dilation) and the inability to obtain accurate timing information due to the presence of the extra instructions (which change the execution history and cause the instruction cache to be polluted).

In the hardware approach, memory references are obtained by either changing the microcode of the processor[14] or by sensing the signals coming from the processor. Both approaches suffer less time dilation than the software approach. However, both approaches are also more costly. The microcode approach is not always possible because a processor's instructions are hard-wired or because there is not enough control store for the modification to be performed. Even if the microcode can be modified, it is likely that instruction timings will be affected. As for sensing the signals coming from the processor, not all the information pertaining to the processor's state may be available.

It was decided that the traces would be acquired with hardware because of the lower time dilation, the non-obtrusive nature of the technique (resulting in greater accuracy), and the desire to acquire timing information. An additional inducement was the availability of a

---

11 A. Borg, R. Kessler, and D. Wall, "Generation and Analysis of Very Long Address Traces", *Proceedings of the 17th International Symposium on Computer Architecture,* ACM, Seattle, WA, May 28-31 1990, pp 270-279.

12 J. Larus and T. Ball, *Rewriting Executable Files to Measure Program Behavior,* Technical Report 1083, Department of Computer Science, University of Wisconsin, March 1992.

13 S. Son, "Senior Project Report: Inline Memory Reference Tracing System", Senior Project Technical Report, Department of Electrical and Computer Engineering, Brigham Young University, May 1991.

14 A. Agarwal, R. Sites, and M Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode", *Proceedings of the 13th International Symposium on Computer Architecture,* IEEE, 1986, pp. 119-127.

hardware trace acquisition system within the university. The heart of the system is the BYU Address Collection Hardware (BACH)[15].

BACH was developed by the Electrical and Computer Engineering department to enable accurate traces to be made of program, operating system, and I/O references. It has been used to show that cache performance can be over-estimated when operating system references are ignored[16].

The trace collection system, shown in figure 3, consists of three parts: the test platform, the address collection hardware (BACH), and the extractor. The platform is the machine which will run the program and whose execution is to be monitored. BACH is primarily a small control unit which regulates the acquisition and placement of memory addresses into a very large buffer. The extractor is another computer which accepts buffers from BACH and stores them on disk to await further processing.

Figure 3: The hardware data acquisition system

Currently there are two supported platforms: a SPARCstation 1+ running SunOS v4.1.2 and a i486 based PC clone running Unixware. Although traces can be obtained using either platform, the SPARCstation was chosen for three reasons. First, the SPARCstation has an external (off-chip) cache that enables virtual memory addresses to be

15 K. Flanagan, B. Nelson, J. Archibald, and K. Grimsrud, "Bach: BYU Address Collection Hardware, The Collection of Complete Traces", *Proceedings of the 6th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, 1992, pp. 128 - 137.

16 K. Grimsrud, J. Archibald, R. Frost, K. Flanagan, and B. Nelson, "Estimation of Simulation Error Due to Trace Inaccuracies", *Proceedings of the 26th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, October 26-28, 1992.

captured. This allows simulations to be performed independent of any paging mechanism. Second, the i486 has a 16 byte prefetch buffer which would have skewed the results of the cache simulation and reduced its effectiveness for non-Intel processors. Finally, the i486 machine was being used to collect traces while benchmarking a commercial database and therefore was not available for use.

The only modifications made to the SPARCstation to enable trace collection were the addition of a daughter card between the CPU and the mother board, and a slowing of the the clock speed to 20 MHz in order to not overwhelm the BACH unit. The daughter card allowed the BACH unit to monitor and log the signals on the CPU pins that comprise the processor's external state. For every change in the external state, the BACH unit recorded an entry containing the number of cycles since the last entry, the address, the data involved, the size of the data read or write, and the processor mode (supervisor or user). In addition, BACH records additional information which is not germane to this discussion. For a more complete explanation of the BACH system, the reader is referred to the design report in reference 15.

2.3 Analysis Technique

The traditional approach to quantifying locality of reference is to count the number of virtual memory page faults that occur during execution[17]. Temporal locality is accounted for since consecutive accesses to the same memory location will not cause a page fault. Spatial locality is also accounted for since accesses to addresses closely surrounding the faulting address will not cause a page fault. In addition, phases of high and low locality can be distinguished by observing the page fault rate or working set size as a function of time.

The primary disadvantage of using the page fault rate as a metric is its large granularity. Typical page sizes are either 1024 or 4096 bytes, while average object sizes are

[17] M. Maekawa, A. Oldehoeft, and R. Oldehoeft, *Operating Systems: Advanced Concepts*, Benjamin Cummings, Menlo Park CA, 1987, Chapter 5.

significantly smaller (around 20 to 64 bytes[18, 19, 20]). Thus it may be difficult for a page fault metric to be sufficiently sensitive to the execution behavior of an object-oriented program. A much finer granularity is needed to insure that intervals of object-oriented locality are measured.

A similar approach which eliminates this problem is to count the number of cache misses during execution. Cache miss rates exhibit the same properties as page fault rates, but at a much finer granularity. Typical first level CPU caches have line sizes (the cache counterpart to virtual memory page sizes) of around 32 bytes and thus approximate the size of an average object. Caches are also included on most computer systems, making cache performance an important consideration. Thus, cache miss rates will be used to measure locality in this study.

The cache configurations of recent high performance microprocessors appear to be converging. As an example, consider the caches from the DEC Alpha[21] processor, the Intel Pentium[22] processor, and the PowerPC 603[23] processor (a joint venture between IBM, Apple, and Motorola). As table 2 shows, each of these processors has separate instruction and data caches. Each of the processors has separate 8 kilobyte instruction and data caches with a 32 byte lines and 2-way set associativity (with the exception of the Alpha processor which has direct-mapped caches of an unreported line size).

---

18 A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983, pg. 659.

19 D. Ungar, "Generation Scavenging: A Non-Disruptive, High Performance Storage Reclamation Algorithm", *ACM SIGSOFT/SIGPLAN Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, May 1984, pp 157-167.

20 G. Krasner (Ed.), *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, Reading, MA, 1983, pp. 94, 96.

21 Z. Cvetanovic and D. Bhandarkar, "Characterization of Alpha AXP Performance Using TP and SPEC Workloads", *Proceedings of the 21st International Symposium on Computer Architecture*, Chicago, IL, April 18-21 1994.

22 *Pentium Processor User's Manual, Volume 1: Pentium Processor Data Book*, Intel, Mt. Prospect, IL, 1994, pg 3-13.

23 B. Burgess, N. Ullah, P. Van Overen, and D. Ogden, "The PowerPC 603 Microprocessor", *Communications of the ACM*, Vol. 37, No. 6, June 1994, pp. 34-41.

Table 2: Representative state of the art caches

	Instruction			Data		
Processor	Size (K)	N-Way	Line (bytes)	Size (K)	N-Way	Line (bytes)
PowerPC 603	8	2	32	8	2	32
Pentium	8	2	32	8	2	32
DECchip 21064 (Alpha)	8	1	†	8	1	†

† Not reported

Based on the information in table 2, the analysis of the locality characteristics of the two programs in this study will be performed with separate instruction and data caches of 8 kilobytes each and a line size of 32 bytes. Both direct-mapped and 2-way set associative caches will be simulated.

Even though they are indicative of the locality of a reference stream, miss rates do not directly indicate system performance. If the speeds of CPU and memory are similar, the cost of a cache miss is small and does not effect system performance. If the speeds are dissimilar, a cache miss will have a detrimental effect. Thus to estimate system performance, memory and CPU cycle times must also be considered.

One parameter that combines the miss rate and cycle times is the effective access time - the average amount of time necessary to resolve a reference. It is defined to be the time to service a cache hit, plus the time to service a cache miss on the average (i.e., miss time scaled by the average miss rate as shown in equation 1).

$$\text{effective access time} = \text{hit time} + \text{miss rate} * \text{miss time} \quad (1)$$

The hit time is assumed to be one processor cycle and is called processor latency. The miss time is assumed to be one memory cycle and is called memory latency. Substituting

these values into equation 1 and dividing both sides by processor latency yields a parameter known as the access multiplier.

$$\text{access multiplier} = 1 + \text{miss rate} * \text{memory latency} / \text{processor latency} \qquad (2)$$

The access multiplier is the average number of processor cycles require to resolve a memory reference. If the ratio of memory latency to processor latency is held constant, then system performance is a function of the miss rate. Conversely, the performance of different system designs can be evaluated if miss rate is assumed.

Table 3 lists memory to processor latency ratios for each of the processors in table 2. An entry is also included for 1 GHz processors expected in the near future. Processor latency is computed as one over the clock speed in hertz. Memory latency is assumed to be 340 ns based on the value reported in reference 21. The use of a single value to compute the latency ratio is possible because the speed of memory is not expected to changing significantly in the future[24]. The results presented in chapter 3 use the range of memory to processor latency ratios from table 3 to estimate the performance impact of adopting an object-oriented approach.

Table 3: Representative system latency ratios

		Latency		
Processor	Clock Speed (MHz)	Memory† (ns)	Processor (ns)	Latency Ratio
PowerPC 603	80	340.0	12.5	27.2†
Pentium	90	340.0	11.1	30.6†
DECchip 21064 (Alpha)	200	340.0	5.0	68.0
Future processors	1000	340.0	1.0	340.0†

† Assuming the memory latency reported for the Alpha

[24] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 1990, pp 426-427.

Chapter 3

Experimental Results

Traces were made of the operating system and test programs while executing each of the input files. Each trace was then filtered to leave only user references. This eliminated a potential bias caused by tracing the operating system, the design and implementation of which are unavailable. Characteristics of the user traces are summarized in section 3.1 and results of the cache simulations are presented in section 3.2.

3.1 Trace Statistics

Traces are frequently summarized by statistical characterizations of the reference stream. To perform this study, over 600 million user and system references (consuming over 1.2 billion CPU cycles) were acquired. Of these, only 311 million user references are of interest in this study. The statistical properties of the traces are shown in tables 4 - 6.

Table 4 shows the percentage of operating system and user references made by the programs. Since only user references are germane, the data shown here will not be extensively discussed. However, it is interesting to note that the C program spent more time in the operating system than did the C++ program. This may be due to differences in the system calls made, but was not investigated further since only user references were of interest.

Table 4: User and OS statistics for the traces

	C		C++	
Input File	% OS	% User	% OS	% User
Compiler	†	†	24.76	75.24
Display	84.05	15.95	77.49	22.51
FileDir	69.01	30.99	39.65	60.35
Files	66.85	33.15	41.58	58.42
Fonts	76.52	23.48	57.07	42.93
Input	84.56	15.44	75.12	24.88
Kernel	91.16	8.84	86.37	13.63
MenuViewer	67.11	32.89	46.61	53.39
Modules	69.04	30.96	44.73	55.27
Oberon	71.45	28.55	39.48	60.52
OCC	61.89	38.11	36.31	63.69
OCE	48.14	51.86	30.67	69.33
OCH	59.81	40.19	35.38	64.62
OCS	66.10	33.90	40.08	59.92
OCT	61.93	38.07	28.11	71.89
Reals	91.97	8.03	88.82	11.18
TextFrames	62.95	37.05	36.10	63.90
Texts	63.86	36.14	29.62	70.38
Viewers	75.18	24.82	47.79	52.21
Mean	70.64	29.36	48.94	51.06
Standard Deviation	11.48	11.48	19.61	19.61

† The OS part of this trace was corrupted by the overflow of a serial port buffer.

Another observation concerns the large overhead for starting a process. This can be seen in both the C and C++ traces as a higher percentage of operating system references during the compilation of short input files such as Display, Input, Kernel, or Reals.

Table 5: Traces statistics for the C program

Input File	References	% Instruction	% Data	Data % Reads	Data % Writes
Compiler	17,468,155	88.03	11.97	10.07	1.90
Display	905,216	87.09	12.91	10.70	2.21
FileDir	4,961,940	86.63	13.37	11.03	2.33
Files	5,665,848	86.71	13.29	10.99	2.30
Fonts	2,058,187	86.53	13.47	11.03	2.45
Input	1,060,361	86.61	13.39	10.97	2.41
Kernel	789,254	86.09	13.91	11.67	2.23
MenuViewer	4,746,409	86.23	13.77	11.27	2.50
Modules	3,535,394	86.99	13.01	10.78	2.23
Oberon	8,162,868	86.85	13.15	11.01	2.14
OCC	10,464,177	87.92	12.08	10.19	1.89
OCE	13,923,362	87.28	12.72	10.50	2.23
OCH	8,131,556	87.06	12.94	10.70	2.25
OCS	5,202,660	86.74	13.26	10.90	2.36
OCT	9,314,356	87.77	12.23	10.34	1.89
Reals	560,641	85.87	14.13	11.67	2.46
TextFrames	15,722,362	86.86	13.14	10.93	2.21
Texts	14,224,160	87.20	12.80	10.69	2.11
Viewers	3,199,647	86.31	13.69	11.30	2.39
Mean		86.88	13.12	10.88	2.24
Standard Deviation		0.59	0.59	0.43	0.19
Total References	130,096,553				

Table 6: Traces statistics for the C++ program

Input File	References	% Instruction	% Data	Data % Reads	Data % Writes
Compiler	27,067,449	83.98	16.02	13.26	2.76
Display	1,218,180	82.29	17.71	14.03	3.67
FileDir	7,328,060	82.26	17.74	14.17	3.58
Files	8,585,780	82.24	17.76	14.14	3.62
Fonts	2,798,234	82.96	17.04	13.44	3.60
Input	1,346,708	82.40	17.60	13.96	3.64
Kernel	604,887	82.23	17.77	14.02	3.74
MenuViewer	6,750,399	82.58	17.42	13.67	3.76
Modules	5,242,678	82.70	17.30	13.93	3.37
Oberon	10,617,578	83.16	16.84	13.59	3.25
OCC	12,310,304	83.89	16.11	13.34	2.77
OCE	23,306,974	82.64	17.36	14.00	3.36
OCH	12,770,470	82.56	17.44	14.01	3.43
OCS	7,427,120	82.86	17.14	13.77	3.37
OCT	13,746,056	83.40	16.60	13.75	2.86
Reals	431,450	81.62	18.38	14.13	4.25
TextFrames	13,819,077	82.89	17.11	13.64	3.47
Texts	21,632,272	83.07	16.93	13.68	3.25
Viewers	4,497,339	81.90	18.10	14.22	3.88
Mean		82.65	17.35	13.86	3.49
Standard Deviation		0.55	0.55	0.26	0.35
Total References	181,501,015				

A comparison of the total number of user references from tables 5 and 6 shows that the C++ program produced 40 percent more references than the C program. Tables 5 and 6 also show that the percentage of instruction and data references exhibit similar characteristics. In both cases, most of the memory references were instruction fetches. A high percentage of instruction references may be due in part to the RISC heritage of the test machine. In the RISC philosophy, operands must be loaded into registers before use. As long as operands can be found in registers, continued program execution occurs without further data references. Contrast this with the CISC tendency to make addressing modes orthogonal to the operation being performed by the instruction. One would expect to have a higher proportion of instruction accesses for a RISC architecture than a CISC architecture. The results support this conclusion.

Another characteristic of both programs is the proportion of data reads and writes. As tables 5 and 6 show, the frequency of read references is four to five times the frequency of write references. Only 2.24 to 3.49 percent of all references write to a memory location. This suggests that increasing the effective speed of memory reads will improve the overall performance of the system for these programs.

The statistics show another important difference in the programs. Comparing the percentages from tables 5 and 6 shows that the C++ program makes 4.23% less instruction references than the C program, while the percentage of data references is higher by the same amount. The percentage of data read references of the C++ program is also lower than the C program by nearly 3%. Overall the C++ program makes 1.25% less read references than the C program. The standard deviations reported for each column indicate that these differences are statistically significant.

3.2 Cache Miss Rates

Table 7 shows that mean cache miss rates for the two programs differ substantially. In fact, the instruction miss rate of the C++ program is 2 to 3 times that of the C program, while the data miss rate is twice as high for the C++ program. The difference in instruction cache performance is similar to that reported by Calder et. al. in reference 6. However, unlike their work which reported no noticeable difference, a comparison of the data cache performance of the two programs indicates that data locality did suffer when adopting an object-oriented style. The increase in data cache miss rate is around 3%, which is nearly as much as the data miss rate of the C program to begin with.

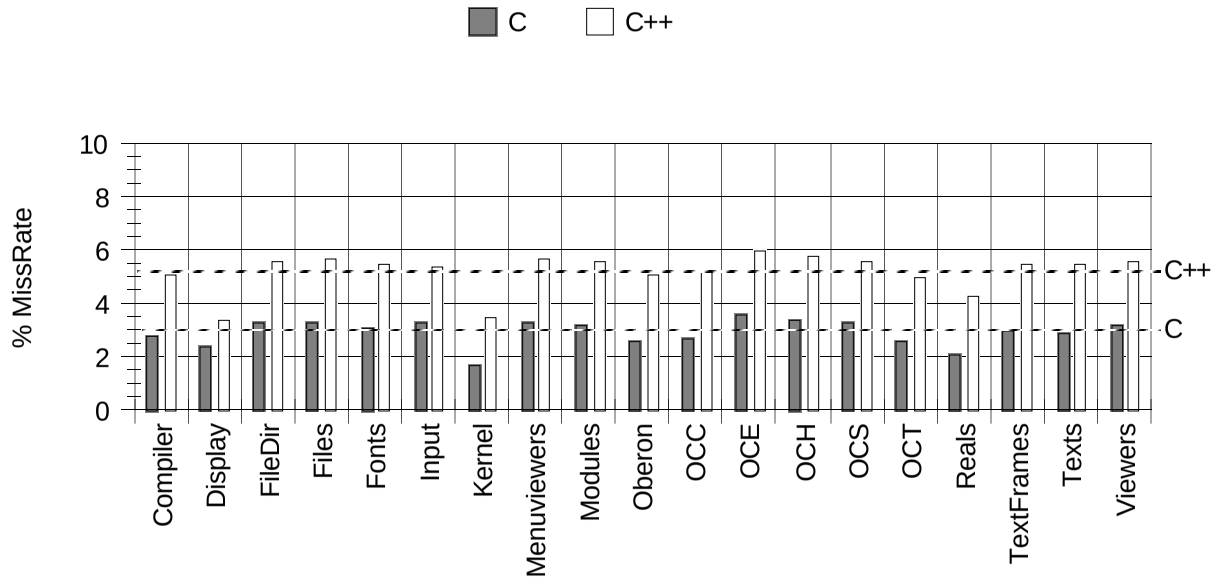
Table 7: Mean cache miss rates in percent

	Associativity	C 1	C 2	C++ 1	C++ 2
Instruction	Mean	2.97	1.26	5.23	3.98
	Std Dev	0.49	0.19	0.71	0.72
Data	Mean	3.50	2.51	6.50	5.24
	Std Dev	1.33	1.39	1.51	1.50

Figures 4 and 5 also show that the cache miss rate of the C++ program was higher than the C program without exception. It can also be seen that the cache miss rate is dependent on the input. For example, the data miss rates of each program are higher for the Compiler traces than for the FileDir traces. In general, the effect of the input file on the miss rate is the same for both programs. This supports the assumption made in section 2.1.3 that the locality of these programs would be dependent on the input.

Figure 4: Miss rates for direct-mapped caches with 32 byte lines

a) 8K instruction cache



b) 8K data cache

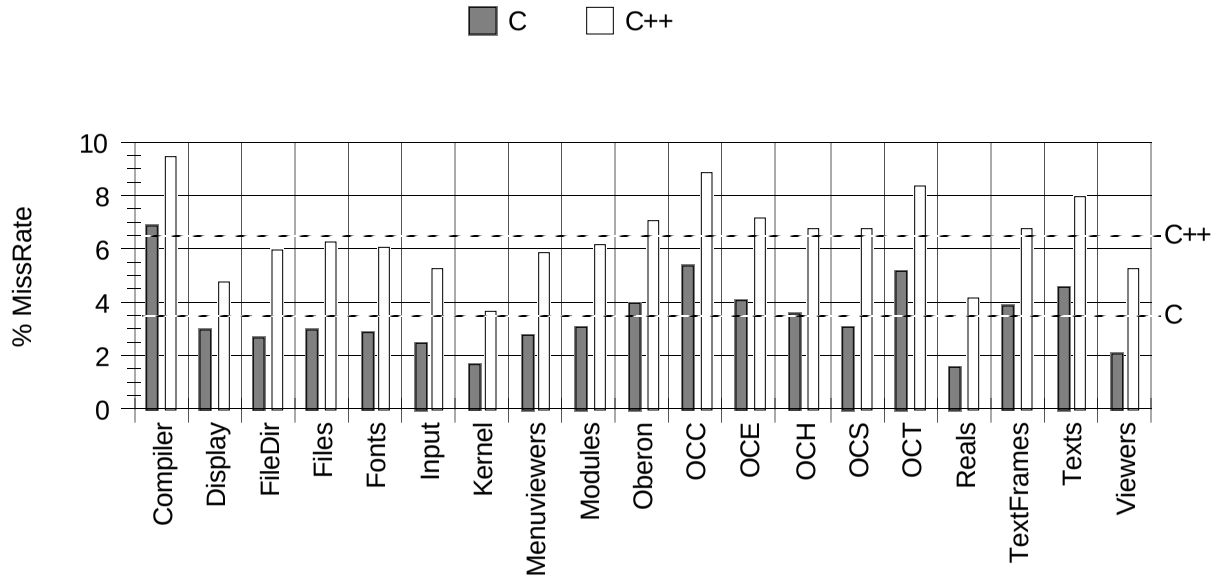
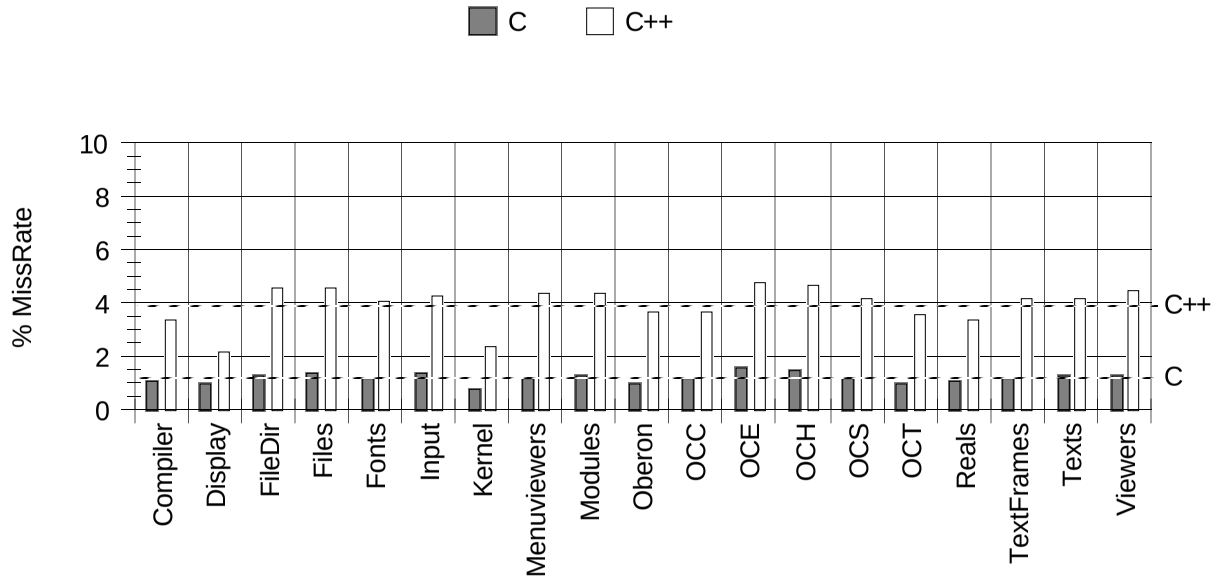
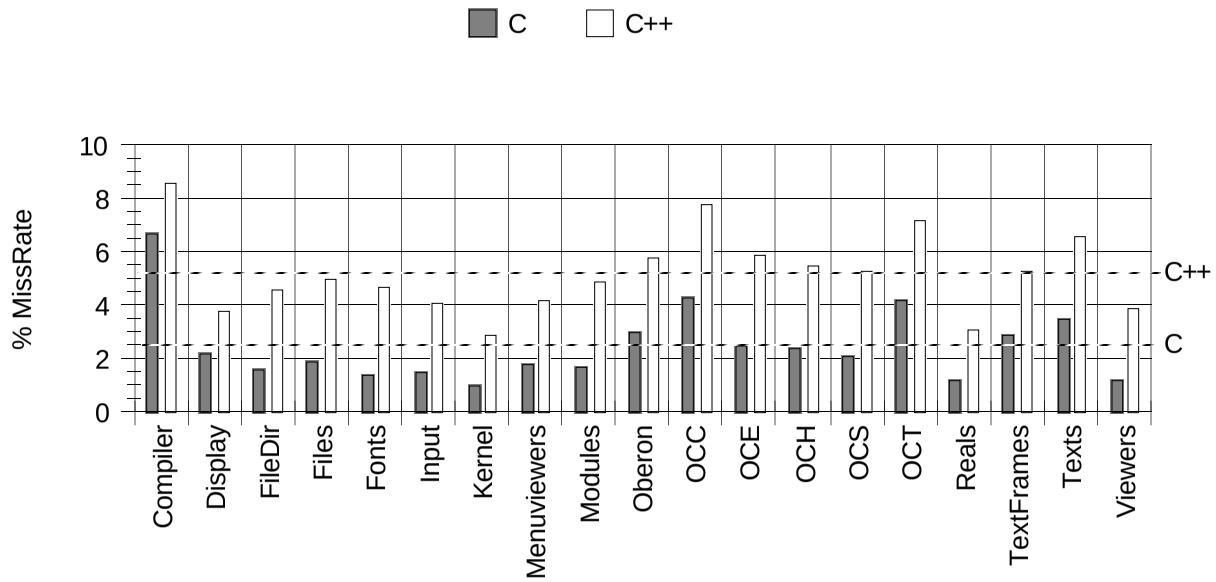


Figure 5: Miss rates for 2-way set associative caches with 32 byte lines

a) 8K instruction cache



c) 8K data cache



Another interesting observation can be made about the change in miss rate that occurs with increased associativity. As cache associativity increases from direct-mapped to 2-way set associative, table 7 reports that the miss rate for the C program decreased from 2.97% to 1.26% (instruction cache) and from 3.50% to 2.51% (data cache). This represents a decrease in the miss rate of 57.6% and 28.3% respectively. The miss rates for the C++ program decreased from 5.23% to 3.98% (instruction cache) and from 6.50% to 5.24% (data cache), representing a decrease of 23.9% and 19.4% respectively. Thus for the caches simulated here, increased associativity does not benefit the C++ program as much as it does the C program. It is not known whether this trend continues for higher associativities or cache sizes other than 8 kilobytes.

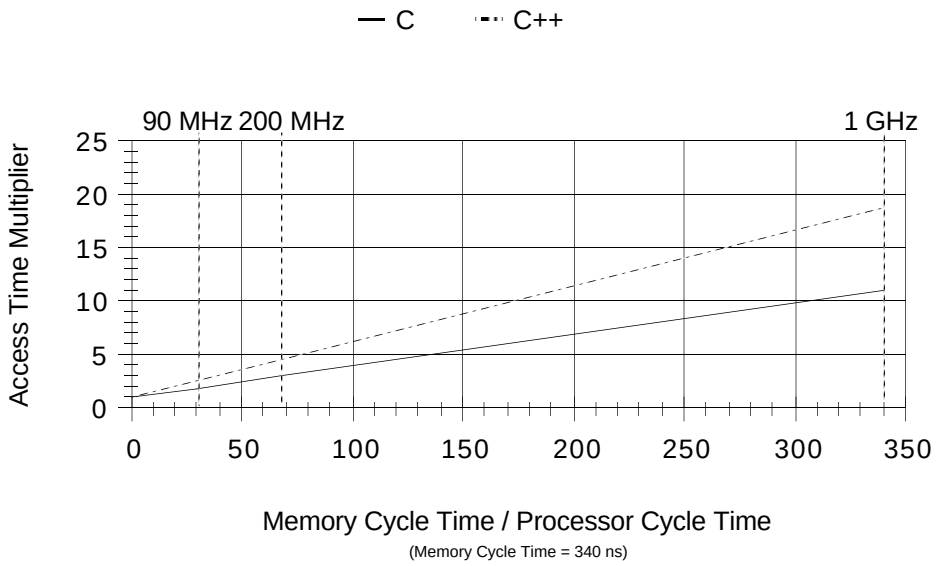
As was discussed in section 2.3, the effective access time and related access multiplier are more indicative of system performance than cache miss rate alone because they include the effect of memory speed upon a cache miss. Figures 6 and 7 compare the access multipliers of the C and C++ programs as a function of latency ratio. Table 8 also presents the results in tabular form. In the following paragraphs, the results for direct-mapped caches will be discussed. The results for 2-way set associative caches are similar.

Table 8: Access multipliers corresponding to figures 6 and 7

		Access Multiplier			
		Direct-mapped		2-way set associative	
	Latency Ratio	C	C++	C	C++
Instruction	30	1.89	2.57	1.38	2.19
	68	3.02	4.56	1.86	3.70
	340	11.09	18.78	5.28	14.52
Data	30	2.05	2.95	1.75	2.57
	68	3.38	5.42	2.70	4.57
	340	12.91	23.09	9.52	18.83

Figure 6: Access multiplier vs latency ratio (direct-mapped, 32 byte lines)

a) 8K instruction cache



b) 8K data cache

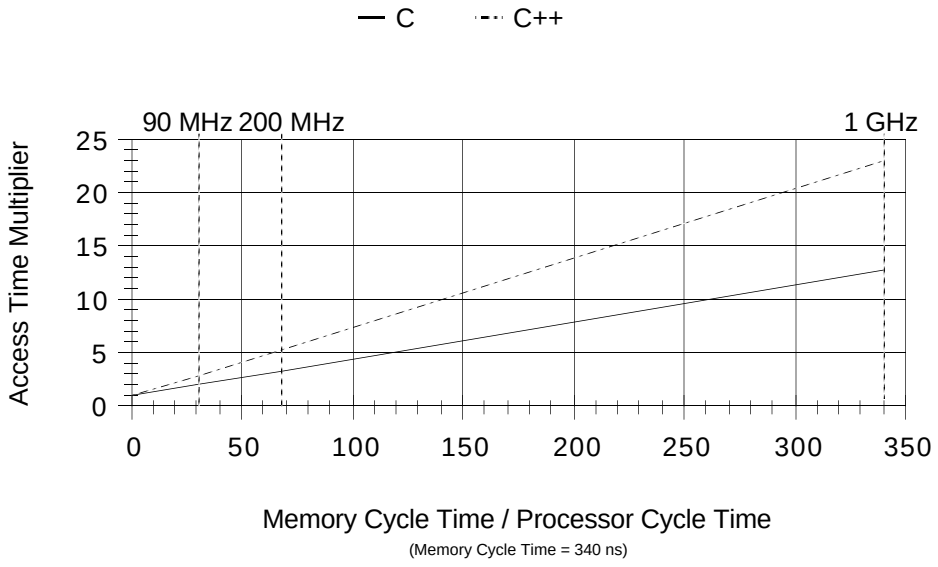
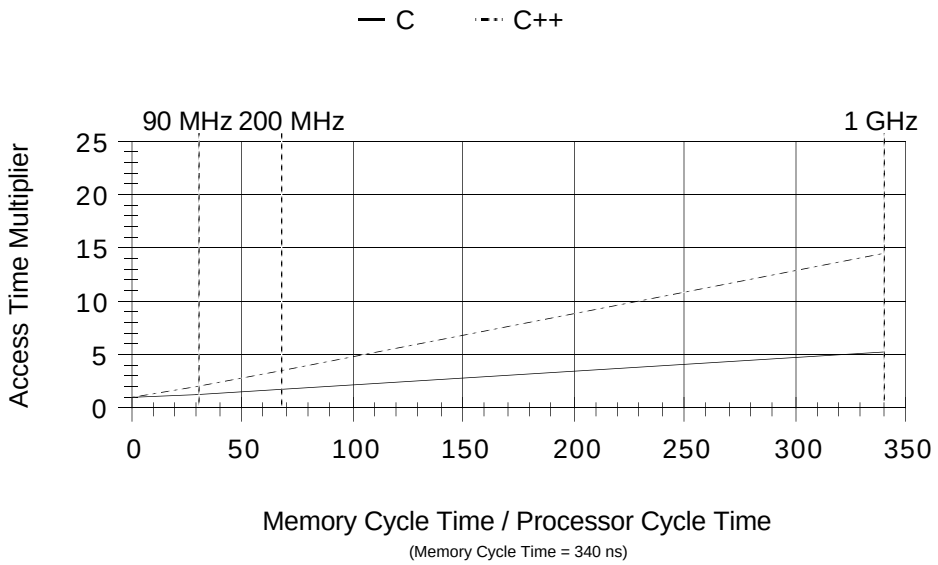
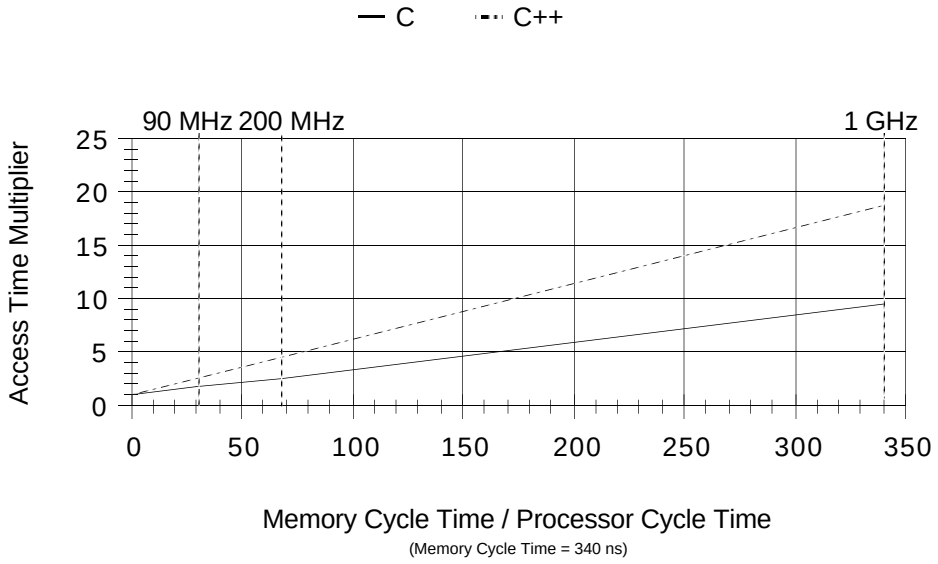


Figure 7: Access multiplier vs latency ratio (2-way set associative, 32 byte lines)

a) 8K instruction cache



b) 8K data cache



Assuming any architecture with a latency ratio of 68 (similar to the Alpha chip) and assuming the same program miss ratios measured in this study, the access multiplier for the C++ program would be 4.56 for instructions and 5.42 for data. The access multiplier for the C program would be 3.02 and 3.38 for instructions and data respectively. Based on the trace compositions reported in tables 5 and 6, the composite access multipliers for the two programs would be 4.71 for the C++ program and 3.07 for the C program. Comparing these access ratios shows that the C++ program takes 53.5% longer to access a memory location on the average. In other words, the C++ program would be expected to take over 50% longer to accomplish the same task. Few applications can afford such a performance penalty.

Although the performance difference is significant for current systems, the results are even more pronounced when processor clock speeds increase. As clock speeds approach 1 GHz, the composite access multiplier of the C++ program becomes 19.53, while the composite access multiplier of the C program is 11.33. The C++ program would be expected to take 72.4% longer. Having to compromise performance by this amount when adopting object-oriented technology is likely to be unacceptable for most applications.

Chapter 4

Conclusions

This study has shown that the locality of reference characteristics of an object-oriented program written in C++ differ significantly from that of a similar program written in C. The C++ program required 40% more references to accomplish the same task. Assuming the latency ratio of a 200 MHz processor and assuming the mean cache miss rates from this study, it is estimated that the C++ program will execute over 50% longer than its C counterpart. As processor speeds increase relative to memory speeds, the effect becomes more pronounced. Such performance penalties are unacceptable for many applications. Future research should concentrate on understanding the reasons for the reduction in locality and investigate ways to minimize the effects of reduced locality on performance.

One fruitful area of research would be to examine how the code generated by a compiler impacts the locality of reference. It would be particularly instructive to know to what extent current implementation techniques for object-oriented concepts like polymorphism affect locality. In addition to compilation, the process of linking needs to be revisited. Previous studies have shown that restructuring a program can have a drastic effect on locality. However, little work appears to have been done on linking object-oriented programs to improve cache performance. Such a study would yield important information with immediate applicability to current object-oriented systems.

Because an attempt was made to exaggerate the effects of adopting an object-oriented style, it is likely that normal design practice will yield programs with better locality than the C++ program used in this study. The degree to which such programs exhibit worse locality than their C counterparts is likely to be dependent on the specific design and

implementation techniques used. It is expected that nearly all C++ programs developed in a production environment should exhibit locality characteristics between the two programs used in this study. However, the effect of specific design and implementation policies on locality remains to be investigated.

Last of all, this study was limited to one application domain - namely that of the compilation of programs. Before the results shown here are assumed applicable to object-oriented programs in general, programs representative of other application domains and implemented in other languages need to be studied.

Appendices

A1 Annotated Bibliography

* A. Agarwal, R. Sites, and M Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode", *Proceedings of the 13th International Symposium on Computer Architecture*, IEEE, 1986, pp 119-127.

Discusses a microcode approach for capturing memory reference traces.

* L. Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer", *IBM Systems Journal*, Vol. 5, No. 2, 1966, pp 78-101.

Discusses the RANDOM, MIN, FIFO page replacement algorithms.

* A. Borg, R. Kessler, and D. Wall, "Generation and Analysis of Very Long Address Traces", *Proceedings of the 17th International Symposium on Computer Architecture*, ACM, Seattle, WA, May 28-31 1990, pp 270-279.

Describes a system for collecting very large traces by adding code to capture addresses at link time. Reference records are written to an OS buffer and a separate high priority process analyzes the data when each buffer is full.

* R. Bunt and J. Murphy, "The Measurement of Locality and the Behavior of Programs", *The Computer Journal*, Vol. 27, No. 3, 1984, pp 238 - 245.

Proposes an approach (Bradford-Zipf distributions) to measuring the intrinsic locality of a program. It is singular in that it attempts to compute locality directly without bias from any particular paging environment. Experimental results are presented to support the technique.

* B. Calder, D. Grunwald, and B. Zorn, "Quantifying Behavioral Differences Between C and C++ Programs", TR698, University of Colorado, January 1994.

Compares a number of C and C++ programs, some of which perform similar (but not identical) functions. The behavior of the C and C++ programs were found to be quite different. The cache performance of the programs was also simulated. The instruction cache performance of the C++ programs suffered substantially. However, the data cache performance of the two sets of programs were essentially the same.

* E. Coffman and L. Varian, "Further Experimental Data on the Behavior of Programs in a Paging Environment", *Communications of the ACM,* Vol. 11, No. 7, July 1968, pp 471-474.

Shows the effect of page size, number of resident pages, and different replacement algorithms on the page fault rate of four programs. One of the primary findings is that increases in resident pages are more beneficial than increases in page size when locality is fine grained. Since objects are small, does this mean that many small pages would be better than fewer large pages?

* R. Courts, "Improving Locality of Reference in a Garbage-Collecting Memory Management System", *Communications of the ACM,* Vol. 31, No. 9, September 1988, pp 1128-1138.

Presents an adaptive memory management algorithm that improves the locality of reference over other forms of memory management based upon garbage collection. Tries to exploit locality of reference to improve garbage collection.

* P. Denning, "Virtual Memory", *ACM Computing Surveys,* Vol. 2, No. 3, 1970, pp 153-189.

Discusses virtual memory, gives a definition of the principle of locality and explains the working set principle.

* P. Denning, "The Working Set Model for Program Behavior", *Communications of the ACM,* Vol. 11, No. 5, May 1968, pp 323-333.

Gives a detailed description of the working set model. Was reprinted in *ACM Computing Surveys,* Vol. 26, No. 1, January 1983, pp 43-48.

* P. Denning, "Working Sets Past and Present", *IEEE Transactions on Software Engineering,* Vol. SE-6, No. 1, January 1980, pp 64 - 84.

An overview of virtual memory and a justification of the working set model.

* D. Embley, B. Kurtz, and S. Woodfield, *Object-Oriented Systems Analysis: A Model-Driven Approach,* Yourdon Press, Englewood Cliffs, New Jersey, 1992.

Describes a formal, model-oriented approach to object-oriented analysis and modeling. An OSA object relationship model was presented in figure 1.

* D. Ferrari, "The Improvement of Program Behavior", *Computer,* Vol. 9, No. 11, November 1976, pp 39 - 47.

Describes several algorithms for restructuring programs. An order of magnitude improvement in page fault rate was observed in some cases.

* K. Flanagan, *A New Methodology for Accurate Trace Collection and its Application to Memory Hierarchy Performance,* PhD Dissertation, Brigham Young University, Department of Electrical and Computer Engineering, 1993.

Discusses the design of the BACH system and its application to caches.

* K. Flanagan, B. Nelson, J. Archibald, and K. Grimsrud, "Bach: BYU Address Collection Hardware, The Collection of Complete Traces", *Proceedings of the 6th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation,* 1992, pp 128 - 137.

The design and implementation of the BYU Address Collection Hardware.

* K. Flanagan, B. Nelson, J. Archibald, and K. Grimsrud, "Incomplete Trace Data and Trace Driven Simulation: A Case Study", *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems MASCOTS,* SCS, 1993, pp 203-209.

Traditional cache simulations can be off by as much as 100 times by neglecting the effect of operating system references.

* K. Grimsrud, *Quantifying Locality,* PhD Dissertation, Brigham Young University, Department of Electrical and Computer Engineering, 1993.

Presents a formal definition of both temporal and spatial locality and their interaction as a 3-D locality surface, discusses a practical algorithm for computing the locality surface, and correlates features of the locality surface to constructs of programming languages. Very thought provoking.

* K. Grimsrud, J. Archibald, M. Ripley, K. Flanagan, and B. Nelson, "Bach: A Hardware Monitor for Tracing Microprocessor-Based Systems", *Microprocessors and Microsystems,* Vol. 17, No. 8, October 1993.

A description of the BACH system implementation for MC68030, i486, and SPARC microprocessors.

* D. Grunwald, B. Zorn, and R. Henderson, "Improving the Cache Locality of Memory Allocation", *SIGPLAN Notices,* ACM, Vol. 28, No. 6, June 1993, pp 177-86.

Presents the locality of reference and performance characteristics of several dynamic memory allocations algorithms evaluated by trace-driven simulation. The characteristics of allocators with bad locality are discussed.

* D. Hatfield, "Experiments on Page Size, Program Access Patterns, and Virtual Memory Performance", *IBM Journal of Research and Development,* Vol. 16, No. 1, January 1972, pp 58 - 66.

Presents results that indicate that for programs that make highly localized use of memory space, increasing (not reducing) the page size increases performance as hardware and software page management overhead decreases. This is compatible with the work done by Coffman and Varian.

* D. Hatfield and J. Gerald, "Program Restructuring for Virtual Memory", *IBM Systems Journal,* Vol. 10, No. 3, 1971, pp 168 - 192.

Describes a technique for re-ordering to increase paging performance.

* M. Hill, "A Case for Direct-Mapped Caches", *Computer*, IEEE, Vol. 21, No. 12, December 1988, pp 25-40.

Points out that although increased associativity gives lower miss rates, the implementation of associativity slows the speed of the cache (thereby negating some of the benefit of higher associativity for first level caches). Also points out that second level caches may benefit from higher associativity. The predicted size at which a direct-mapped cache becomes faster is based on miss rates substantially lower than those found in this study.

* J. Larus and T. Ball, *Rewriting Executable Files to Measure Program Behavior*, Technical Report 1083, Department of Computer Science, University of Wisconsin, March 1992.

Discusses the capture of program behavior by rewriting executables. This technique was used by Calder et. al. in their comparison of the locality characteristics of a set of C and C++ programs.

* E. Lau, *Performance Improvement of Virtual Memory Systems*, UMI Research Press, Ann Arbor, Michigan, 1982.

Discusses ways to improve performance of virtual memory systems. Gives FFT as an example of a program with spatial locality, but not temporal locality (accesses subsets of locations in close spatial proximity, but those subsets change quickly with time).

Also suggests (on page 89) that forward branches are more likely than backward branches. This is contrary to what I remember seeing (although I cannot find any references).

* A. Madison and A. Batson, "Characteristics of Program Localities", *Communications of the ACM*, Vol. 19, No. 5, May 1976, pp 285 - 294.

Discusses the concept that programs exhibit a hierarchy of localities. Gives a definition of bounded locality intervals and an algorithm for calculating them. Would be interesting to see if bounded locality intervals could be determined from memory reference traces rather than the high-level Algol traces used in their work.

* J. Schemer and G. Shippey, "Statistical Analysis of Paged and Segmented Computer Systems", *IEEE Transactions on Computers*, Vol. EC-15, No. 6, December 1966, pp 855 - 863.

Applies statistical analysis to analyze paged and segmented memory via simulation. Defines "page reference distribution functions" for reference strings of essentially random, sequential, or exponential behavior.

* A. Smith, "Two Methods for the Efficient Analysis of Memory Address Trace Data", *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January 1977, pp 94-101.

Discusses two techniques for reducing the size of long reference strings, along with an analysis of the inaccuracy resulting from the reduction.

These techniques may not be necessary anymore due to the dramatic increase in memory size and computational power of workstations. For example, the traces taken in support of the work reported in this paper were rather long (over 600 million references), yet they did not require reduction.

* S. Son, "Senior Project Report: Inline Memory Reference Tracing System", Senior Project Technical Report, Department of Electrical and Computer Engineering, Brigham Young University, May 1991.

Discusses a software-based approach to trace generation. The assembly output of a C compiler is annotated to log the information necessary to construct the desired trace. Post-processing removes the offset bias of the logging code.

* J. Spirn, *Program Behavior: Models and Measurements,* Operating and Programming Systems Series, Elsevier, New York, 1977.

Presents a general overview of modelling program behavior and reviews the LRU, OPTIMAL, and WORKING SET models.

* J. Spirn, "Distance String Models for Program Behavior", *Computer,* Vol. 9, No. 11, IEEE, November 1976, pp 14 - 20.

Describes Belady's Lifetime Function, one of whose parameters can be used to gauge the locality of reference. Also contains plots of mean working set size and page fault range vs. window size.

* J. Spirn, and P. Denning, "Experiments with Program Locality", *Proceedings of the AFIPS Fall Joint Computer Conference,* Vol. 41, 1972, pp 611 - 621.

Discusses two types of locality models: the intrinsic model (assumes locality results from a program's internal properties) and the extrinsic model (defines locality from observable properties reference strings). Proposes experimental criteria to test the ability of extrinsic measurements to reflect the current intrinsic locality

A2 Availability of Source Code and Traces

The source code and trace data from this study is available for further research. Please contact the Department of Computer Science, Brigham Young University, Provo, UT 84602.

A Comparative Study of the Locality Characteristics of an Object-Oriented Language

Mark K. Gardner

Department of Computer Science

M.S. Degree, November 1994

ABSTRACT

Modern computer systems are designed to exploit locality of reference to improve performance without increasing cost. Due to the recent adoption of the object-oriented paradigm, the locality characteristics of object-oriented programs have not been firmly established. This study estimates the locality of an object-oriented C++ program by comparing its cache performance with that of a C program performing the same task. The C++ program was found to have a significantly higher cache miss rate. Decreased locality indicates decreased performance. This effect will become more pronounced as processor speeds increase.

COMMITTEE APPROVAL:

Aurel Cornell, Committee Chair

Scott N. Woodfield, Committee Member

David W. Embley, Graduate Coordinator